Chapter 9: System structure


Processes can access shared data objects; they may share an entire address space
they may execute in separate but segmented address spaces in which segments may be shared
if the processes’ address spaces are completely disjoint, or are such that they cannot share data segments, we must consider how processes can share information without having access to shared data objects (shared segments & shared OS: typisch voor UNIX)
fig: multi-threading => shared address space (=> Java: virtuele machine: kan opgevat worden als process met meerdere draden)

9.3 Sharing the OS

 Fig1: In order that a process can easily make system calls to request system services, it is often arranged that the OS code and data occupy the same part of every process’s address space, typically one half of it. UNIX is structured in this way (shown in fig). when a process is executing in user mode it may be separate from all other processes (as in UNIX). When it requests system service by making a system call, it enters a region which is shared with all other processes. OS code accesses shared OS data / original UNIX design with three segments at user level, one for code (text segment) and two for data. Although the text segment could be shared, it was not possible to share any data areas.
fig2: arrangement of the virtual address space (MIPS architecture)
half the virtual address space is for user code and data, half for the OS. Much of the OS must be resident in memory, ie, it must not be swapped out to backing store and need never be moved around in memory once loaded. It is possible to make these parts of the OS directly mapped onto physical memory, thus avoiding the need for address translation for references to them. The address translation table is a scarce resource and this is a good way of conserving it. (als stukken steeds op zelfde plaats in geheugen terechtkomen: geen mapping, want je kent fysieke geheugenplaats al; niet alles wordt gecached omdat dit niet altijd toegelaten is; software-matige indeling, niet hardware-matig!)

9.5 Requirements for process interaction

 

Fig3: (a) processes may share same address space; here we have a multi-threaded process; the process is the unit of resource allocation and threads are the unit of execution and scheduling and the active entities which need to communicate
(b) processes may run in separate address spaces on the same machine. Here: heavyweight process model where the process is both the unit of resource allocation and the unit of scheduling and execution; user processes may run in separate address spaces on the same machine. System processes on this machine may share an address space. It might be that user processes become system processes when they make system calls and enter the OS
(c) processes may run in separate address spaces on different machines
((a) A en B voeren zelfde operatie quasi simultaan uit => probleem: ze zien allebei nog 1 zetel beschikbaar en claimen die allebei => KRITISCHE SECTIES (b) beide processen draaien op 1 processor, A wordt ondebroeken, B gaat verder met net dezelfde operatie als A => zelfde probleem als in (a); mogelijke oplossing voor (b): zeggen dat er geen interrupts mogen zijn zodat B pas start nadat A helemaal afgewerkt is (brengt uiteraard weer andere gevolgen met zich mee…); geen opl voor (a))

* *Processes may need to cooperate to carry out some task* => one process may need to make a request for service of some other process and eventually wait for the service to be done
it may also arise that one process reaches a point where it must ensure that some other process has carried out a certain task before continuing
general requirement for **synchronization**. Cooperating processes need to sync with each other and to do this need to be able to WAIT for each other and SIGNAL to each other
* *Processes may compete for exclusive use of services or resources* => competing processes need to WAIT to acquire a shared resource and need to be able to SIGNAL to indicate that they have finished with the resource

Support for process interaction
fig3 shows interacting processes, and the implementation of support for he interactions, in some of the process placement diagrams. The term inter-process communication is used in the figure to include the types of interaction.

(process kan zowel echt process als draad binnen process betekenen
WAIT & SIGNAL (zie ook Ch4): interrupts => sync tov hardware; hier sync tov andere processen)

9.6 Types of process interaction

One-to-one
**static configuration** of interactions between individual processes. These processes send their data to or receive commands from other known mgmt. processes. The interactions which will take place are known in advance; in particular, the names of the processes involved are known.

Any-to-one
multiple clients of a single server
in a given interaction one client invokes one (possibly well-known) server. The server cannot know which potential client will next invoke it and needs to be able to WAIT for ‘anyone’ to make a request. The server process name can be made known in advance to all clients.

One-to-many
used to notify a set of interested parties that an event has occurred
**broadcast**: if a communication is sent out to ‘everyone’
**multicast**: communication is sent to a specific set of recipients

Any-to- (one of) many
a given service might be provided by a number of anonymous processes. A client needs to request service from any free service rather than one particular named process

Many-to-many via shared data
processes may be able to share writeable data. Any number of processes can communicate through a shared data structure.

9.7 A process interaction

9.7.1 Problems when processes share data in memory
if processes share an address space, the system designer may make use of shared data objects to effect cooperation of manage competition for resources. If shared data objects can be written as well as read by concurrent processes, problems can arise
 (a) two processes executing on two processors of a multi-processor. Both read the value of *unbooked*-*seats* into a register (either from their caches or from main memory) and find the value is 1. They each proceed to book it.
(b) two processes running on a uniprocessor wih preemptive scheduling. Process A reads the value of *unbooked*-*seats* into a register and is then preempted in favour of process B. the saved state of A includes the values stored in the processor register. B reads the value of *unbooked*-*seats* into a register, finds there is one seat left and books it. When A runs again its saved state is set up in the processor registers; if finds there is one seat left and proceeds to book it. => problem comes from uncontrolled access to shared data, because implementation of the high-level language statement requires several machine language instructions

9.7.2 Problems when processes do not share data in memory
a system in which processes do not shara memory requires information to be passed between processes as well as the WAIT- SIGNAL synchronization requirement. It is possible to support the transfer of data around the system from process to process.
problems associated with this:
\* naming: the processes involved must know each other’s names in order to communicate
\*the time to copy data between address spaces or to move it across a network
\*the time to context-switch between processes running in separate address spaeces and the number of context switches involved

9.7.3 Granularity of concurrency
the size of a process, ie, the number of operations it comprises, is system- or app-dependent. It might be appropriate to use a separate process to carry out operations on each element of a vector or array => **fine-grained concurrency**, typically used to achieve a highly parallel solution on a special-purpose architecture such as a vector or array processor
**coarse-grained concurrency** : the use of one process for each user job in a time-sharing system.
(geen shared data als je stricte afscheiding wilt => bescherming)

(1) ‘in’ waar ik gegevens mag toevoegen
‘out’ waar gegevens eruit gehaald worden, schuift steeds verder op
five-slot: ik kan maar 5 dingen i/d buffer steken en dan is hij vol (5 ingangen)
(2) producer & consumer: coöperatief => ze werken samen; toch ergens competitief: variabelen die ze beiden bijhouden (counts) maar geen concurrenten in zin van producer enkel in, consumer out